Minimization Package Review

Jim Kowalkowski
Marc Paterno
CD/CEPA/APS/SLD

21 April, 2004

Contents

1 Introduction

This review considered the MINIMIZATION library, a facility for numerical minimization of functions. We looked at the most recent version of the code[1]. We considered both design and coding issues, concentrating primarily on a user's interaction with the library, and secondarily on internals.

Because of time constraints, this document contains detailed analysis of few classes; some classes are not mentioned at all. We concentrated our comments on a few classes in order to more fully explain the *kinds* of modifications we believe would be valuable. The same sort of modifications could be made in other classes as well.

1.1 Goals of this Review

The goals for this review were to answer the follow questions:

- *Is this package easy to use?* Are the examples and documentation clear enough? Are the interfaces of the user-level classes cohesive and complete?
- *Does the class decomposition make sense?* Does each class have a single, clear purpose? Are the names of classes, functions, *etc.* helpful for the user's understanding?

[1]"Most recent" in this case means the code at the head of the CVS repository as of 15 April, 2004. See §6.1 for comments on code versioning.

- *Is the library extensible?* Can users add new functionality with reasonable effort?
- *Is the library maintainable?* Can the author add new basic functionality with reasonable effort? Will maintenance programmers be able to support the code when the main author has moved on to another project?

2 Overview

The MINIMIZATION package provides an object-centric library for the numerical minimization of functions. In broad terms, it aims to provide, and extend upon, the functionality provided by the venerable Fortran package MINUIT.

The package goals include ease of use and ease of extensibility. A user who needs to apply the provided minimization techniques to his own problem should be able to do so easily. A mathematically sophisticated user who wishes to add a new minimization algorithm to the library should be able to do so with a minimum of coding effort.

3 Major Concerns

In this section, we summarize our most important concerns with the package. Later sections go into greater detail on these topics, and sketch possible solutions.

3.1 Major User Interface Concerns

3.1.1 Ease of Use

In experimenting with the library by writing our own `Function` subclass, we found that it was difficult to figure out why the minimization wasn't converging[2]. The tools for determining the state of the algorithm were insufficient; it was inconvenient to control the process of minimization. In one sense, too much was hidden from the user.

At the same time, a great deal of detail was "in our face". For example, we could not merely ask for the current best estimate of the minimum of the function; first, we had to check to see that the value was valid, and perhaps tell the `Problem` to evaluate the function once, and then we could ask for the value of the function at the current point. It would be far better to have a function to get the value of the function at the current point, and for this function to take care of the details of what is necessary to do so.

3.1.2 Useful Extensions to the Package

We see a need for an interactive tool. The interactive mode of MINUIT is one of its greatest strengths. It seems that the current version of MINIMIZATION is not so far away from allowing the production of an interactive tool that uses a `Problem` to perform minimization under user direction.

We believe it is best to prepare the library to support the development of such a tool, but we do *not* believe that the tool should be built into this library. Such a tool will benefit from

[2]It turns out the reason for the failure was a simple coding error in our implementation of the `Function` subclass. We expect such errors to be unfortunately common, and so it is important that the package provide sufficient tools to allow the user to investigate the behavior of the package.

graphical support; a number of different well-known techniques exist for separating an interactive user interface from a core library. One such way is to allow intermediate returns from `Problem::minimize` when specified conditions are met. Such conditions should include (but not necessarily limited to):

- stop at a state change, and
- stop after n iterations.

Additional interesting intermediate conditions should be identified and supported. During an intermediate return, the user should be able to view the algorithm state and take other actions similar to the ones supported by MINUIT, such as fixing or freeing parameter, or scanning the function by varying a single parameter.

This package appears to be almost ready for a command line interpreter to be added. We did not get a chance to explore whether or not the `MinimizationState` contained all the necessary information to support such functionality.

3.2 Major Design Concerns

We believe many of the problems discussed in this document are the result of insufficient modularity and cohesiveness of the classes in the system. We believe each class should have a clearly defined purpose, and provide an interface sufficiently complete to fulfill that purpose, and free of features unrelated to that purpose. We find that too many of the classes in the package are not treated as *classes*; their instances are treated as buckets of data, with the data being manipulated externally by other code. This is why we describe the library as object-centric, rather than object-oriented.

There is insufficient decomposition of functionality within classes. In many places within the `Algorithm` subclasses, we find complicated state information within a single function. We also find that many of the methods do not do one task; they do different tasks based on the state of other objects in the system. Many of the methods were found to contain switch statements, which almost always tends to couple a system tightly together and also add more branching than is probably needed. These functions should be decomposed for better clarity. As it now exists, the code presents significant maintenance challenges.

3.3 Major Coding Concerns

We are concerned with the relatively large number of `switch` statements in the code; the class `RectilinearDomain` alone contains thirteen. Widespread use of `switch` constructs in C++ is often the sign of insufficiently coherent classes. We believe that this is the case with many of the `switch` statements in this package.

Frequently, we find that the `switch` statement is choosing a behavior based upon a state variable. These state variables seem to be indicators of unformed classes. In general, we would strongly prefer to see such logic resolved using classes and virtual functions.

3.4 Major Physical Design Concerns

Our major physical design concerns are with the build system. We believe use of ISOCXX and SOFTRELTOOLS should be eliminated, because they will decrease the size of the user community able to easily take advantage of this library.

3.5 Redundancy in Interfaces

Many of the classes and functions provide multiple ways of accomplishing the same task. We see no reason to support more than one way to do something and recommend removing alternative methods. The resulting smaller interface is easier to understand, to document, and to test for correctness. Examples are: ways to set all parameter values or just a single one, and ways to request a specific one or all parameters values.

4 Design Recommendations

Our design recommendations can be divided into two broad categories:

- recommendations regarding the *user's* view of the package, and
- recommendations regarding the *maintainer's* or *extender's* view of the package.

In the first group we include those who code their own `Function` subclasses. Authors of new `Algorithms` or `Domains` we include in the second group.

We consider recommendations of the first sort to be more critical. We believe that user acceptance of the package will be determined largely by the user's experience of the interface of the package.

We believe recommendations of the second category to be important for the long-term viability of the package. If it is too difficult to maintain or extend, the package will stagnate.

4.1 Overview

Many of the classes expose their internal details by providing either accessor methods or public data members. Some of the classes actually expose pointers to internal data that can be manipulated by a user. This lack of encapsulation leads to excessive coupling, and hinders ease of use.

Several of the classes have methods that only apply in only some states and need to be ignored under other conditions. This can cause coupling problems, and makes it much more difficult to understand how the classes function. It also leads to state checking of the object flavor before making use of it—a form of dynamic cast. Some of the classes in this situation are

- `Function`
- `Problem`
- `Domain`

Is this done to prevent the proliferation of classes? It is likely that the functionality of *several* classes is packaged in a few classes, with extra methods and state existing to discover the true type buried in the design. Some refactoring of the internals would increase the maintainability of the code significantly.

4.2 Problem Class

Currently most of user interaction revolves around the `Problem`. For this reason, we have concentrated much of our effort in analyzing the interface of `Problem`.

4.2.1 Resource Management

There is no problem with having more than one `Problem` object at a time, but instance of the class `Problem` may not be copied. There appear to be two reasons: the use of `auto_ptrs` as data members, and the difficultly of copying objects known only by an abstract interface (cloning). It does *not* seem that there is an innate aspect of the concept of a "problem" that makes is nonsensical to think of copying one[3]. We urge that the restriction on copying `Problems` be removed. One disadvantage of not allowing copying is that it becomes difficult to use a `Problem` as a data member of another class. This can be trouble in a program where the user does not have control over the main routine or where another library may want to use this library.

We see no reason to use `auto_ptr` in the *implementation* of `Problem`; see later comments about the use of `auto_ptr` in the *interface* of `Problem`. A `Problem` could instead contain bare pointers, managed appropriately.

There is a significant amount of literature that discusses solutions to the cloning problem—using wrappers and a class's own copy constructor, for example. In this case, the domain is sufficiently restricted that it seems such solutions may be more than is needed. Requiring that all subclasses of `Algorithm`, *etc.* have a member function `clone()` which returns a newly made instance of the same type is not burdensome.

4.2.2 Interface Problems

Because `Problem` is the user's main view of the MINIMIZATION package, we believe it is of the highest importance to have its interface be as clean as possible.

The `Problem` class contains an unhappy mix of methods: general runtime configuration parameters (iteration counts, tolerance), readout of results, commands to force internal state changes (`computeInverseHessian`), commands to control behavior, and accessors of internal data. Some tasks are made needlessly tedious, and require the user to invoke sequences of calls known only from examples or documentation. An example is getting the current function value. This can only be done if function `valueValid` returns `true`. Another example is getting the scale when an algorithm like `Simplex` is used. It looks like you get back something nonsensical. We would prefer to see each of these access functions perform whatever calculations are needed to produce the correct answers, and to return that answer. If, for some reason, it is impossible to calculate a quantity, this should be regarded as an exceptional situation—and an exception should be thrown.

We suggest reorganization of the interface of `Problem` along the following lines.

4.2.2.1 Functionality Belonging in `Problem`

As a general guideline, we view `Problem` as a Facade, which directs the interaction of the `Algorithm`, *etc.* and which provides for the user a simple interface for the performance of most interesting minimization tasks. We believe the `Problem` should provide the user's view of those functions which will be available in all subclasses of the base classes with which `Problem` interacts. It should *not* provide functions with which the user should *not* interact; if internals of the MINIMIZATION package need access to such things, they should be obtained elsewhere.

The functions which we believe should remain in `Problem` are:

- `setTolerance`

[3]We contrast `Problem` with a class that represents an external resource, *e.g.*, a memory mapped file. It makes sense for an object representing a memory mapped file to be uncopyable, because the semantics of copying such an object would be problematic. No such conceptual difficulty appears in the copying of a `Problem`.

- `tolerance`
- `setPoint`—we suggest renaming this to `setGuess` for clarity.
- `point`—we suggest renaming this to `currentLocation` for clarity.
- `minimize`
- `functionValue`—we suggest renaming this to `currentValue`, and modifying the semantics so that it does not fail silently. If there is not an up-to-date cached value of the value of the function at the current estimated location of the minimum, the function should be evaluated and the value returned. This function should only fail if it turns out to be impossible to evaluate the function at the current best estimate of the minimum. This being an exceptional condition, an appropriate exception should be thrown.
- `order`—we suggest renaming this to `numParameters` for clarity.
- `functionCallCount`
- `computeInvHess` and `getInvHessState`—we recommend significant reorganization of the interface to this information. Users may want access to the value of the inverse Hessian; such access should be granted through the interface of `Problem`. They may also need to know the quality of the estimate, and furthermore they might or might not want to obtain a value only if one is cached. We recommend two functions:
  - `InvHessianState getInverseHessian(InverseHessian& result)`, which will perform the calculation if a value is not already cached, and which will throw an appropriate exception if it fails in such a calculation, and
  - `InvHessianState getCachedInverseHessian(InverseHessian& result) const`, which accesses a cached value.

  Each of these functions returns an `InvHessianState`, which indicates the status of the matrix being return in the *out* argument `result`.
- `setAlgo`—we recommend modifying this function to be a member function template:

  ```
  template <class ALG>
  ALG* setAlgorithm(ALG a);
  ```

  The behavior of this function would be to copy the given object of type `ALG` (which would compile only if the given object was an instance of a subclass of `Algorithm`), and to return a (non-owning) pointer to the object retained by the `Problem`. The user can hold onto this pointer to obtain access to any specialized interface of its concrete class.
  We are not worried about inefficiency in the copying of the `ALG` object, because such copying is done rarely. If this is a serious concern to others, we would then propose:

  ```
  template <class ALG>
  ALG* setAlgorithm(std::auto_ptr<ALG> p);
  ```

  which would allow the `Problem` to take ownership of the same `ALG` instance, and to return its address to the user via a pointer which does not convey ownership the object.
  Note also the change in name. We see little to benefit in the slightly shorter `setAlgo`, rather than the clearer `setAlgorithm`. We believe names like the latter are easier to read and remember.
- `setTerm`—we recommend the same sort of modifications here as we did for `setAlg`, templatizing the function and spelling out the name as `setTermination`.

It might be useful to add additional functions to drive the minimization; one example might be `takeStep`, which is to drive the `Algorithm` through one iteration of its minimization procedure, and to then return.

It might be useful to add a second function to evaluate the function being minimized, allow the user to specify the location at which to evaluate the function: `evaluateFunction(const Point& p)`.

We strongly urge the addition of a constructor template to allow the user to create a `Problem` instance that contains the correct `Algorithm` subclass upon initialization, rather than making the user construct a `Problem` with an inappropriate `Algorithm` type, followed by replacement of that `Algorithm` with a new one of the correct type.

We also urge the addition of a constructor template to allow the user to create a `Problem` instance with the desired `Termination` type.

```
template <class FUNC,
          class ALG=Minimization::Migrad,
          class TERM=Minimization::StandardTermination>
Problem(FUNC f, ALG a, TERM t);
```

4.2.2.2 Functionality Belonging in `Algorithm`

The functions `setStrategy` and `strategy` are unclear in meaning and potentially dangerous, because they have meaning only in the context of how a particular `Algorithm` subclass interprets the (integer) value. This functionality, if needed, should be moved to the particular `Algorithm` subclasses, and the user can interact with the `Algorithm` directly if needed.

4.2.2.3 Functionality Belonging in `Termination`

Several of the functions presently in `Problem` should be moved to those `Termination` subclasses which support such notions, or to the Termination base class if *all* termination criteria must support such the functionality. The functions in question are:

- `functionCallLimit`,
- `functionCallDefaultLimit`,
- `setFunctionCallLimit`,
- `incrementFunctionCallLimit`

4.2.2.4 Functions to be Removed

The `problemState` method should be removed. This method can only cause trouble because it exposes all the internal data of the system to be manipulated at will.

The functions `paramName` and `setParamName` should be removed. Nothing in the system uses these features. They may be appropriate for a command line or GUI tool that provides a front end to the MINIMIZATION package.

The accessors that return pointers to internals of the `Problem` as pointers to base classes should be removed. They should be replaced with member templates which return explicit derived types. Below is an example of one to retrieve the algorithm.

```
template <class ALG>
ALG* algorithm();
```

4.3 Domain Class

We considered what a user would need to do to write a `Domain` to represent a (2-D) circle. The user needs to write several functions:

- `computeInternalPoint`, which transforms a point from polar to rectangular coordinates. This is simple.

- `computeExternalPoint`, which transforms a point from rectangular to polar coordinates. This is also simple.
- `computeInternalScale`, which transforms a point and uncertainty in rectangular coordinates to a point an uncertainty in polar coordinates.

  We find it distracting that uncertainty estimates are called "characteristic scales". This may be common jargon for implementers of minimization algorithms, but it not the most convenient naming for less-expert users.

  On each invocation, this function is expected to indicate that the first parameter (corresponding to the radius r) is not periodic, and the second parameter (corresponding to the angle θ) *is* periodic. Why is this to be indicated on *every* call? Must not the same values be returned each time from this `Domain`?

  We did not have sufficient time to complete our investigation of the Domain class. From the analysis we did complete, it seems possible for the package to be simplified by removing Domain as an abstract class, leaving only `RectilinearDomain`. Any additional review of the package internals should consider this possibility.

4.4 Point and its Isomorphs

The `Point` class gives a interface that better matches the user's need that that of a vector. It is unclear why the vector should be exposed by the `Vector()` method. An acceptable method might be `const double* asSimpleArray() const` to get at the values as a simple C style array.

In debug mode, out of bounds checking is done directly in the `Point` class. The purpose of the vector `at()` method is to perform this function.

There are many classes in the system that are the same (or just about the same) as the `Point` class. The only difference appears to be the name of the class.

- `Hessian`- included 2D access plus order as member
- `SecondDerivatives`
- `Gradient`
- `Point`
- `Correlations`
- `CharactericScale`
- `InternalHessian`- same as `Hessian`
- `InternalScale`
- `InternalPoint`
- `InternalSecondDerivatives`
- `InternalGradient`

We certainly acknowledge that all of these are different concepts and that allowing the type system of C++ find incompatibilities is valuable. However, the current arrangement bloats the package and makes it more difficult for the user to make use of the system (more header files to plow through to find out what is available in each of the classes). We strongly recommend getting rid of all these, except for one, and using `private` inheritance to generate the unique types.

4.5 Termination Class

We suggest the function `finished` be renamed `isFinished`, to clearly indicate that this is a *query* as to whether or not the termination criterion is met, as opposed to a declaration that the termination criteria *has* been met.

4.6 ProblemState

`ProblemState` is used in the program as a fancy common block. Almost all the user definable classes (*e.g.* `Algorithm`) manipulate it directly even though many of the entries do not make sense to all classes. In some configurations, it appears to be possible that some of the entries are meaningless. If possible, this classes should be broken up into parts of closely related functionality needed by specific methods.

4.7 "hasThis" and "canDoThat" methods

The `Domain` and the `Function` classes have methods that are valid only when other methods indicate that they are valid. We find this to be a poor design choice, evident in the need to construct a large number of case statement to frequently check capabilities. A better decomposition into cohesive classes would simplify this sort of logic and result in a more maintainable system.

4.8 Contour Class

The `Contour` class exists to calculate contours of the function being minimized. This is an important feature when the minimizer is being used for a fitting problem, and perhaps in other cases as well.

Once a `Contour` instance is created, the only thing one can do with it is to print it to an `ostream`. While this has some utility, we believe that a GUI interface to the MINIMIZATION package would provide a much superior graphic representation of the contours. To support such a GUI, the relevant behavior of `Contour` would be to produce an evaluation of the function on a grid—in the same manner as that used to produce the ASCII artwork plot now made by printing a `Contour` object.

5 Coding Recommendations

There is a calculation of the machine epsilon which is not needed—the values wanted are available in:

- `std::numeric_limits<long double>::epsilon()`
- `std::numeric_limits<double>::epsilon()`
- `std::numeric_limits<float>::epsilon()`

If quantities derived from these values are needed, such quantities should be constructed starting from these elements.

5.1 AlgorithmUtility

While looking into the importance of the class `Domain`, we came across some remarkable numerology in `invHessian`, which resides in `AlgorithmUtility`. Many explicit magic numbers appear in the code, in lines such as:

```
if (p.parameterIsPeriodic[i] && d > .5) d = .51;
```

which actually appears in two places in this function. The proliferation of such magic numbers contributes to making the code *extremely* hard to understand. We strongly urge their removal.

5.2 UserFunction

If two ways are going to be supported for supplying function arguments, we recommend using `const Point&` and a C style array `const double*` instead of a `std::vector<double>`.

6 Physical Design Recommendations

6.1 Versioning

There is no version tagging in the CVS repository, making it hard to know what version of the code is being used. We recommend that tagged releases of the package be established, so that users can clearly identify which version of the code they are using. The version tagging provided by CVS rtag is appropriate, and most convenient for existing Fermilab experiments.

6.2 Use of ISOCXX

The makefiles in each of the build directories make use of ISOCXX. ISOCXX was created to support the use of pre-standard compilers. Such compilers are used increasingly rarely, and thus ISOCXX is no longer of much benefit. Since the use of ISOCXX in this package limits its ability to be used in environments *not* supported by ISOCXX, we believe that it should be removed from the MINIMIZATION package.

6.3 Use of SOFTRELTOOLS

While several current experiments at Fermilab use SOFTRELTOOLS, it is not used outside the lab. We believe it is necessary to support the use of the MINIMIZATION package by the experiments using SOFTRELTOOLS, but that it is *not* beneficial for the MINIMIZATION package itself to be built using SOFTRELTOOLS. Because the code size of the package is small, a few standalone *makefiles* might suffice as a “build system.” If more is deemed necessary, we suggest the use of the GNU AUTOMAKE toolchain.

6.4 Optimization of DEBUG Builds

Because of the switches overridden in *src/GNUmakefile*, even a DEBUG build is made with *-O3* optimization. This makes it nearly impossible to actually use the debugger to determine what the program is doing. A much less aggressive optimization level—perhaps none—should be used in the debug build.

7 Documentation

We believe most users will look at examples before documentation, so we did so as well. Because of this, it is critical to get these examples in the best possible shape. We found several problems with the examples.

7.1 Simple Example Needed

The examples contain much extraneous code; code that does not directly apply to understanding how to best use the library to solve a problem. The code for a random number generator and the setup function that produces the dataset do not contribute to learning about the minimization package. Most of this code is clutter and just in the way. A simple, easily interpreted function definition and simple and small already-prepared dataset will make the example much more clear:

Listing 7.1: example1.cc

```
// 5 samples, each containing x, y, and uncertainty in y
static double x[5][3] = { {1,2,.1}, {3,4,.1}, {5,6,.1},
                          {7,8,.1}, {8,9,.1} };
class MyFunc : public Minimization::Function {
  explicit MyFunc(double dataset[5][3]) : Function(3) : x_(dataset) { }

  double operator()(const Minimization::Point& a) const
  {
    double sum=0.;
    for(int i=0; i<5; ++i)
    {
      // y = a * cos( x + b ) + c
      //    x = x_[i][0]
      //    y = x_[i][1]
      //   dy = x_[i][2]

      double v = x_[i][1] - ( a[0] * cos( x_[i][0] + a[1] ) + a[2] );
      sum += v*v / x_[i][2]*x_[i][2];
    }
    return sum;
  }

  double (*x_)[3];
};
```

Now you can just do the simple main like in examples 1 and 2 and it clear what is going on.

There is not a good, clear example that matches what the experimental physicists do. The coding style in examples 4 and 5 make it difficult to quickly know what is being fitted and what is input data. Examples 4 and 5 are must too similar; just setting a different algorithm can be done as a commented out line in example 4. There should be an example of a domain manipulation. If `Termination` is expected to be used by developers, then a simple example of a custom `Termination` object should be included.

8 Conclusion

We were encouraged by how quickly we were able to add our own `Function` class to the system and make use of it. The improvements listed in this document will make this an easier task for those less familiar with C++ and the concepts within this library. Of particular importance are the method name change suggestions, a cleanup of the examples, and a better organization of the `Problem` class.

We targeted producing three sample applications: (1) a χ^2 minimization, (2) adding a genetic algorithm to do the minimization, and (3) one driving the library from a foreign language. Unfortunately we only had time to produce the first of these. As for the genetic algorithm, this technique is sufficiently different from the current algorithms that is it difficult to know the relevance of classes like domain. We kept the third application in mind while solving (1), and see no fundamental problems.

This review concentrated mostly on external aspects. We suggest another review be scheduled after all such aspects are addressed that looks further into refactoring the internals to make this package easier for a person that is not a primary author to maintain.